Compilación de programas lógicos que utilizan la negación por falla.

Una extensión de la máquina abstracta de Warren [1]

Alejandro J. García[2] Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Instituto de Ciencias e Ingenieria de la Computación
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Av.Alem 1253 - (8000) Bahía Blanca - ARGENTINA
FAX: +54 (91) 883933 Tel.: +54 (91) 20776 (int. 208)
e-mail: ccgarcia@criba.edu.ar grs@criba.edu.ar

Palabras claves: Programación en lógica, máquina abstracta de Warren.

Resumen

Con el objetivo de disponer de un lenguaje que capture los aspectos del razonamiento rebatible, se ha desarrollado un *lenguaje de programación en lógica rebatible* como una extensión de la programación en lógica convencional, el cuál fue presentado en los trabajos [12] y [1]. Actualmente se está desarrollando una máquina abstracta para los programas lógicos rebatibles, y luego se construirá un interprete en base a esta máquina abstracta, utilizando la semántica definida en [12]. De esta forma, se podrá obtener un sistema de características mas adecuadas para la programación de sistemas basados en conocimiento. Este trabajo constituye el primer paso en ese sentido: extender la máquina abstracta de Warren para permitir utilizar la negación por falla.

La *máquina abstracta de Warren*, o Warren Abstract Machine (WAM) ha sido aceptada como un estandar para la implementación de Prolog, y por este motivo se la ha utilizado como punto de partida de este desarrollo. Como los programas lógicos rebatibles utilizan la negación por falla, el objetivo principal de este trabajo es extender la máquina abstracta de Warren para incluir un conjunto de instrucciones que permita utilizar la negación por falla como un operador predefinido en el lenguaje. Además, se presenta el desarrollo de un compilador que traduce un programa lógico a instrucciones WAM, un soporte de ejecución para administrar la memoria de la arquitectura abstracta, y por último un intérprete que ejecuta el conjunto de instrucciones WAM.

---

[1]Financiado parcialmente por la Secretaría de Ciencia y Técnica, Universidad Nacional del Sur.

[2]Becario de Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), República Argentina

1 Introducción

La programación en lógica es uno de los principales exponentes de la programación declarativa. Aunque se la ha utilizado como herramienta de representación de conocimiento, presenta limitaciones para modelar el razonamiento común. En los últimos años se han desarrollado extensiones de la programación en lógica (véase por ejemplo [5, 8, 7]), las cuáles incorporan algunos aspectos del razonamiento rebatible (donde conclusiones previas son refutadas ante la presencia de mayor información). Estas extensiones han enriquecido el lenguaje de programación pero aún presentan algunos problemas (ver [2]).

Con el objetivo de disponer de un lenguaje que capture los aspectos del razonamiento rebatible, y solucione los problemas de dichas extensiones, se ha desarrollado un *lenguaje de programación en lógica rebatible* como una extensión de la programación en lógica convencional, el cuál fue presentado en los trabajos [12] y [1]. Los *programas lógicos rebatibles* están formados por dos tipos de cláusulas: las *cláusulas de programa extendido* y las *cláusulas de programa rebatible*, separando de esta forma el conocimiento estricto (seguro), de la información rebatible (tentativa).

Actualmente se está desarrollando una máquina abstracta para los programas lógicos rebatibles, y luego se construirá un interprete en base a esta máquina abstracta, utilizando la semántica definida en [12]. De esta forma, se podrá obtener un sistema de características mas adecuadas para la programación de sistemas basados en conocimiento. Este trabajo constituye el primer paso en ese sentido: extender la máquina abstracta de Warren para permitir utilizar la negación por falla, y presentar el desarrollo de un compilador para programas lógicos con este tipo de negación.

La *máquina abstracta de Warren*, o Warren Abstract Machine (WAM) es una máquina virtual que consiste de una arquitectura de memoria y un conjunto de instrucciones diseñadas especialmente para la ejecución de Prolog. Actualmente la WAM ha sido aceptada como un estandar para la implementación de Prolog, y por este motivo se la ha utilizado como punto de partida de este desarrollo. Varios trabajos han descripto en detalle los principios de diseño de la WAM, entre los cuáles "*Warren Abstract Machine – A tutorial reconstruction*" [6] de Hassan Aït-Kaci, es uno de los más claros y concisos.

El diseño original de al WAM no posee definidas instrucciones para programas lógicos que incluyan negación. Como los programas lógicos rebatibles utilizan tanto la negación por falla, como la negación clásica, el objetivo principal de este trabajo es extender la máquina abstracta de Warren para incluir un conjunto de instrucciones que permita utilizar la negación por falla como un operador predefinido en el lenguaje.

La WAM constituye sólo una parte de la implementación de un lenguaje de programación en lógica. Para poder utilizarla se necesita un compilador que tome como entrada un programa lógico y genere código ejecutable sobre la WAM. Otro objetivo de este trabajo es presentar el complemento de la obra de Hassan Aït-Kaci, esto es: un compilador

que traduce un programa lógico a instrucciones WAM, un soporte de ejecución para administrar la memoria de la arquitectura abstracta, y por último un intérprete que ejecute el conjunto de instrucciones extendido de la WAM.

No está dentro de los objetivos de este trabajo la descripción detallada de la WAM, sin embargo, se introducirán algunos conceptos a fin de que pueda ser entendido el diseño del compilador, y las nuevas instrucciones que manejan la negación por falla. Este trabajo puede resultar además una buena aproximación al estudio de la WAM, ya que presenta el paso previo al uso de la máquina abstracta: la compilación de los programas lógicos. Se espera que el lector posea un conocimiento básico de programación en lógica, principalmente en lo que se refiere a la semántica operacional de Prolog, los mecanismos de unificación y backtracking (ver [9]).

2 Concepción básica de la WAM

En la presente sección se presentará la concepción básica de la WAM, y se asumirá que el lector posee cierto conocimiento de programación en lógica (detalles sobre la WAM pueden encontrarse en [14, 15, 6] y sobre la programación en lógica en [9]).

En el lenguaje de la programación en lógica, un *término* es una *variable*, o una *estructura* de la forma $f(t_1, \ldots, t_k)$ $(k \leq 0)$ donde cada t_i es un término. Si $k = 0$, f se llama *constante*. Un predicado atómico es una estructura de la forma $p(t_1, \ldots, t_k)$, donde p es el nombre del predicado y cada t_i es un término. Cada t_i se llamará *argumento* del predicado p. Para diferenciar las variables de los demás elementos, éstas se denotan con un identificador que comienza con una letra mayúscula.

Un *cláusula de programa normal* [9] es una cláusula de la forma $P \leftarrow Q_1, \ldots, Q_n.$ donde el consecuente P es un predicado atómico, y el antecedente $Q_1, \ldots, Q_n$ representa una conjunción de predicados atómicos. Todas las variables de una cláusula de programa están implicitamente clausuradas universalmente, y además cada Q_i $(1 \leq i \leq n)$ puede estar precedido por el operador *not* de la negación por falla. Las cláusulas de programa no son implicaciones de la lógica clásica; deben tomarse como reglas de inferencia, y no poseen un valor de verdad. Su semántica informal [9] es: si el antecedente es verdadero, entonces el consecuente es verdadero. En una cláusula como la anterior, se llamará *cabeza* a P, y *cuerpo* a $Q_1, \ldots, Q_n$. Un *programa normal* es un conjunto finito de cláusulas de programa normal, en el cuál se tienen los siguientes elementos:

- Hechos: predicados atómicos P que establecen una aserción.
- Consultas: cláusulas de la forma $\leftarrow Q.$ que denotan una *meta* a resolver.
- Reglas: cláusulas de programa $P \leftarrow Q_1, \ldots Q_n.$ $(n \geq 0)$.
- Respuestas: Sustitución resultante de la unificación de toda variable de una consulta.

Observese que de acuerdo al significado que tienen las cláusulas de programa, dada la regla "$P \leftarrow Q_1, \ldots, Q_n$.", "$P$" será un hecho, si se satisfacen todas las consultas Q_i. Por lo tanto las reglas están formadas por los dos elementos anteriores, esto es, "consultas" Q_i que al resolverse implican un nuevo "hecho" P. Esta forma de ver las reglas permite que todo el desarrollo de la WAM se haga sólo sobre consultas y hechos. El siguiente ejemplo se utilizará para mostrar los principios sobre los cuáles trabaja la WAM.

Ejemplo 1 :
Dado el programa

p ← r.
r.
← p.

el código WAM correspondiente es el siguiente:

```
p/0 :  just_me    p  |  r/0 :  just_me    r  |  ?:  just_me
       allocate   ←  |         allocate    .  |      allocate   ←
       call r/0   r  |         deallocate     |      call p/0   p
       deallocate .  |         return         |      deallocate .
       return        |                        |      return
```

En cada columna se muestra el código y a su derecha (a modo de comentario) la cláusula de la cuál proviene, intentando dejar cada elemento de la cláusula junto a la instrucción que generó. El símbolo "?" se utiliza para representar el comienzo del código de la consulta. □

La WAM funciona básicamente de la siguiente manera: en la memoria está almacenado el código WAM del programa lógico, y de la consulta. La máquina abstracta deberá ejecutar las instrucciones generadas, comenzando en la instrucción rotulada con "?". La ejecución se desarrollará secuencialmente excepto cuando se encuentren instrucciónes de salto (como `call`), las cuáles modificarán el flujo del control. La ejecución comienza con una consulta Q (← p. en el ejemplo), ella es la encargada de cargar datos en los registros y en la memoria (HEAP) y luego llamar al predicado que corresponde. Si existe una cláusula C cuya cabeza unifique con Q (en el ejemplo "p ← r."), entonces se ejecutan las consultas que corresponden al cuerpo de C. El proceso continúa hasta que las consultas invocadas se unifican con hechos (la consulta tiene éxito), o alguna de las consultas no encuentra una cláusula para unificar (la consulta falla).

2.1 Conjunto de instrucciones

Las instrucciones de la WAM pueden agruparse según su función, de la siguiente forma:
a) **Instrucciones de control**: `call`, `return`, `allocate`, `deallocate`, `just_me`, `try_me_else`, `retry_me_else`, `trust_me` y `fin_consulta`. Estas instrucciones, como se

vió en el ejemplo 1, son las encargadas de llevar el flujo de control del programa. La instrucción `call` es la encargada de buscar la definición de la cláusula que unifique con la consulta; `return` simplemente devuelve el control a la instrucción siguiente de la llamada; `allocate` y `deallocate` son las encargadas de apilar y desapilar los registros de activación que contendrán información que debe preservarse mientras se resuelven las consultas del cuerpo de la cláusula. Las demas instrucciones poseen información para el mecanismo de backtracking y se verán con mas detalle en la sección 3.1.

b) Instrucciones para consultas: `put_var`, `put_value`, `put_structure`, `put_const`, `set_var`, `set_value` y `set_const`. La ejecución de un programa está direccionada por consultas, son ellas las encargadas de cargar los registros o la memoria con valores nuevos (put), o fijar su valor con algún valor anterior (set). Existe una instrucción especializada para cada tipo de elemento: variables, constantes o estructuras. Más adelante se verá en detalle cuando se utiliza cada una de ellas.

c) Instrucciones para hechos: `get_var`, `get_value`, `get_structure`, `get_const`, `unify_var`, `unify_value` y `unify_const`. Los hechos (o las cabezas de reglas) son los encargados de tomar los datos que las consultas han dejado en algún lugar de la memoria. Para ello se tienen dos tipos de instrucciones para cada elemento de programa, las instrucciones que toman datos (get), y las que intentan unificar dos elementos (unify).

2.2 Variables, registros "X" y registros de argumento "A"

El manejo de las variables y la unificación, son temas centrales dentro de la definición de la máquina abstracta, ya que: (1) la computación dentro de un programa lógico se produce en su mayoría a través de la unificación de variables de una consulta con estructuras de un hecho, y (2) todas las apariciones de una variable dentro de una cláusula toman el mismo valor (clausura universal implícita de las cláusulas). Para esto es necesario establecer un mecanismo de comunicación entre una cláusula que intenta resolverse y el hecho que espera ser llamado. Este mecanismo es parecido al pasaje de parámetros de los lenguajes imperativos.

Los valores de las variables son almacenados en la memoria. Según el uso que se les dará a estos valores, se almacenan en el Heap, en registros (denotados con X_j), o registros de argumentos (denotados con A_i). Los registros X_j son los que "conservan" el valor de las variables durante la ejecución de una cláusula. En una cláusula cada variable tiene un único registro asociado. En cambio, los registros de argumento A_i, sólo sirven para pasar la infomación de una consulta a un hecho y por lo tanto hay un registro por cada argumento en la consulta.

Ejemplo 2 :

(i) En la consulta " ← p(V).", hay un registro X_1 para la variable "V" y un registro A_1 para el único argumento "V".

(ii) En la consulta " ← p(U,V,f(U,V,W),U,a)." hay tres registros: X_1 para la variable

"U", X_2 para la "V", y X_3 para la "W". En cambio, hay cinco registros de argumento: A_1 para la "U", A_2 para la "V", A_3 para la estructura "f(U,V,W)", A_4 para la tercera aparición de "U", y A_5 para la constante "a". □

A diferencia de las variables, las constantes no tienen un registro X asociado, ya que pueden tomar un único valor: ellas mismas. Las estructuras, como no son funciones, se comportan de manera análoga a las constantes. No obstante, como se verá en el ejemplo 3, las constantes y estructuras, que no están adentro de otra estructura, sí tienen asociado un registro de argumento A_i.

Ejemplo 3 :
El programa

p(X,X) ← r(X,X).
r(a,Y).
← p(U,V).

tiene el siguiente código WAM

```
p/0:  just_me          p
      allocate         (
      get_var X1 A1    X,
      get_value X1 A2  X)←
      put_value X1 A1  r(X
      put_value X1 A2  ,X
      call r/0         )
      deallocate       .
      return

r/0:  just_me          r
      allocate         (
      get_const a A1   a,
      get_var X1 A2    Y
      deallocate       )
      return           .

?:    allocate         ← p(
      put_var X1 A1    U,
      put_var X2 A2    V
      call p/0         ).
      deallocate
      fin_consulta
```

Observando los lugares que ocupa la instrucción `call`, puede verse que las consultas colocan datos en lugares de memoria con instrucciones "put", mientras que los hechos los obtienen con instrucciones "get". Debe recordarse que las cláusulas se interpretan como un hecho, seguido de varias consultas. □

Además de los registros anteriores, la WAM posee los siguientes registros especializados: "PC" (program counter), "H" (puntero al Heap), "E" (puntero al Stack de entornos), "B" ,"HB" (punteros al Stack de backtracking), y "TR" (puntero al trail).

3 Compilación y generación de código

Cada cláusula de un programa normal puede compilarse por separado, si cierta información es guardada en una tabla de símbolos que perdure hasta que se halla compilado todo el programa. El compilador que se ha construido es de una sola pasada, y utiliza dos tablas de símbolos, una cuyo tiempo de vida es la compilación de una cláusula, (que se

llamará *temporaria*), y otra que perdura incluso durante la ejecución (que se llamará *permanente*). El analizador léxico, es el encargado de leer el archivo del programa y entregar al analizador sintáctico uno a uno los elementos encontrados. Para la construcción del analizador sintáctico se utilizó un análisis recursivo descendente, ya que las cláusulas de programa están definidas recursivamente. El analizador sintáctico recorre una a una las cláusulas del programa, almacenando la información obtenida en las tablas de símbolos.

La tabla de símbolos temporaria se utiliza para procesar la información interna de cada cláusula, y por lo tanto tiene la siguiente información: identificador encontrado, categoría (variable, constante, estructura, predicado, etc.), número de registro X, número de registro de argumento A, aridad (en el caso de los predicados y estructuras), y la instrucción WAM asociada al elemento.

La tabla de símbolos permanente, en cambio, se utiliza para almacenar los datos del programa. Por cada cláusula se almacena: el identificador del predicado de la cabeza, su aridad, la posición donde comienza el código de dicha cláusula, e información para el backtraking. Además como no hay una restricción en la longitud de un identificador, todo identificador de estructura o constante es guardado en esta tabla a fin de optimizar el uso de la memoria.

Como casi todos los elementos de una cláusula tienen asociado una instrucción WAM (ver ejemplo 3), las instrucciones se van generando y guardando en la tabla temporaria, a medida que se va compilando la cláusula. Una vez que se ha completado su compilación, se carga en memoria el código WAM, la tabla temporaria es vaciada, y se continúa con la compilación de la siguiente cláusula del programa.

La compilación de una cláusula está divida en la compilación de la cabeza (un hecho), y luego, si existen, la compilación de cada una de las consultas del cuerpo. Al detectar el nombre del predicado de la cláusula, este se guarda en la tabla temporaria, se generan las instrucciones `just_me` y `allocate` y luego se siguen compilando (recursivamente) los argumentos del predicado. Al terminar de compilar los argumentos, se conoce la cantidad de estos, y por ende la aridad del predicado, la cuál es guardada en la tabla de símbolos. La compilación de cada una de las consultas del cuerpo de la cláusula es similiar a la de la cabeza: se guarda el identificador de la consulta en la tabla temporaria, se genera el código de los argumentos, y luego con el identificador de la consulta que está en la tabla, se genera la instrucción `call`. Una vez que se ha terminado de compilar el cuerpo de la cláusula, se generan las instrucciones `deallocate` y `return`.

La compilación de estructuras es muy similar a la de un predicado atómico, con la diferencia que no debe generarse código de control. Si se está compilando un hecho, y se encuentra una estructura, se genera la instrucción `get_structure`, y luego se compilan sus argumentos. Si la estructura es encontrada dentro de una consulta, se procede de igual manera pero se genera la instrucción `put_estructure`. Para las constantes hay cuatro posibilidades: si está en un hecho es `get_const` o `unify_const` según esté afuera o dentro de una estrucura, y si en cambio está en una consulta será `put_const` o `set_const`.

La mayor dificultad está en generar una instrucción WAM para una variable. En el momento de encontrar una variable, el compilador debe distiguir: si está compilando un hecho o consulta, si la variable está dentro de una estructura o no, y si es la primera aparición o no. En función de esa información procede como lo indica el siguiente algoritmo:

Si está compilando una consulta, entonces:
  Si es la primera aparición de la variable, entonces:
    Si está dentro de una estructura, entonces la instrucción es `set_var Xj`
    Si no está dentro de una estructura, entonces la instrucción es `put_var Xj Ai`
  Si no es la primera aparición de la variable, entonces:
    Si está dentro de una estructura, entonces la instrucción es `set_value Xj`
    Si no está dentro de una estructura, entonces la instrucción es `put_value Xj Ai`

Si no está compilando una consulta (*i.e.*, está compilando un hecho), entonces:
  Si es la primera aparición de la variable, entonces:
    Si está dentro de una estructura, entonces la instrucción es `unify_var Xj`
    Si no está dentro de una estructura, entonces la instrucción es `get_var Xj Ai`
  Si no es la primera aparición de la variable, entonces:
    Si está dentro de una estructura, entonces la instrucción es `unify_value Xj`
    Si no está dentro de una estructura, entonces la instrucción es `get_value Xj Ai`

3.1 Backtracking

Cuando un predicado P tiene más de una definición (*i.e.*, existe un conjunto de cláusulas p_i para definir a P), si una de las definiciones p_i falla, se debe probar con otra p_j (*i.e.*, se realiza backtraking). En el conjunto de instrucciones de la máquina abstracta, hay tres instruciones especializadas en manejar el backtraking. Éstas son: `try_me_else`, `retry_me_else` y `trust_me`. Si por ejemplo un predicado posee cuatro cláusulas que lo definen, entonces el código de la primera comenzará con la instrucción `try_me_else L` que se interpreta como "pruebe ésta cláusula y sinó, pruebe con la que está en la dirección L". El código de la segunda y la tercera comenzarán con `retry_me_else L`, y el de la última con `trust_me`.

Las cláusulas que definen un predicado pueden estar intercaladas con otras cláusulas, e incluso estar en diferentes archivos. Por lo tanto cada vez que encuentre una nueva defincón de un predicado, el compilador deberá modificar el código compilado para colocar la instrucción de backtracking que corresponda. Para esto se utiliza información almacenada en la tabla de símbolos permanente. Cada vez que se compila una cláusula, su código es almacenado en la memoria encabezado por la instrucción `just_me`. Esta

instrucción no estaba incluída en el conjunto de instrucciones original de la WAM, se la incluyó para indicar cuando un predicado tiene una sola definición.

En la tabla de símbolos temporaria, toda cláusula tiene como primera instrucción `just_me`. Al generar código, si es la primera aparición de una cláusula para definir el predicado, entonces se deja `just_me`, y se actualiza la información de la tabla de símbolos permanente. Pero si ya había una definición anterior de este predicado (esto es, ya figuraba en la tabla de símbolos permanente como predicado definido), entonces se debe corregir su código. Pueden ocurrir dos cosas: (1) que sea la segunda cláusula en aparecer, entonces se reemplaza la instrucción `just_me` por `try_me_else`, o (2) que ya existan dos o más cláusulas que lo definen, y entonces se reemplaza en la última, `trust_me` por `retry_me_else`. En ambos casos la primera instrucción del código de la nueva cláusula es `trust_me`.

4 Inclusión de la negación por falla

El objetivo principal de este trabajo es definir una extensión de la máquina abstracta de Warren que permita ejecutar programas normales que utilizan la negación por falla (naf) [3]. De esta forma, las consultas podrán estar precedidas por el operador *not* de la negación por falla, cuya semántica operacional es la siguiente: una consulta " ← *not* q." tendrá éxito, si la consulta " ← q." falla.

Al conjunto de instrucciones WAM se agregaron dos nuevas instrucciones `set_naf` y `end_naf`, las cuáles marcarán en el código el comienzo y fin de una consulta "negada" (precedida por *not*). Al momento de la compilación cuando se encuentra el símbolo *not*, se genera la instrucción `set_naf`, luego se compila la consulta igual que antes, y al final del código de ésta, se genera la instrucción `end_naf`.

Ejemplo 4 : A continuación figura un programa con "not" y su código WAM correspondiente:

p ← not r.
← p.

```
p/0 :  just_me     p      ?:  just_me
       allocate    ←          allocate    ←
       set_naf     not        call p/0    p
       call r/0    r          deallocate  .
       end_naf     .          return
       deallocate
       return
```

[3] En inglés "negation as failure"

La ejecución de la consulta " ← p." tendrá exito ya que la consulta " ← not r." no puede satisfacerse. □

Se incorporaron nuevos registros especializados para manejar la negación por falla: "N", "C", "PC_NAF", "H_NAF", "E_NAF", "B_NAF", "HB_NAF", y "TR_NAF" El registro "N" inicialmente está en cero, y mantiene el número de operadores *not* que se encuentran sin resolver; mientras que "C" indica que la proxima vez que se ejecute la instrucción `call`, ésta debe realizar una tarea adicional por ser un llamado precedido por el operador *not*. Ambos registros son modificados al ejecutarse `set_naf`, cuyo código es el siguiente:

```
PROCEDURE set_naf;
BEGIN
  N := N + 1;
  C := TRUE;
END set_naf;
```

Los demás registros son utilizados para salvar (mientras dure la ejecución de la consulta negada), los valores de los registros de la WAM "PC", "H", "E", "B","HB", y "TR". Como a su vez, la cláusula que unifica con la "consulta negada" puede contener otra consulta precedida por el operador *not*, el contenido de los registros "PC_NAF", "H_NAF", "E_NAF", "B_NAF", "HB_NAF", y "TR_NAF" debe guardarse en el "Stack". Esto lo realiza la instrucción `call`.

```
ROCEDURE Call;
BEGIN
IF C   {La llamada esta precedida por ''not''} THEN
  BEGIN
    Salvar_en_el_Stack(PC_NAF, H_NAF, E_NAF, B_NAF, HB_NAF, TR_NAF)
    PC_NAF:=PC; { Actualiza los registros *_NAF }
    H_NAF:=H;
    E_NAF:=E;
    B_NAF:=B;
    HB_NAF:=HB;
    TR_NAF:=TR;
    C := FALSE; { y C vuelve a ser falso }
  END; {Hubo un ''$not$'' antes}
    .. aqui continua el codigo de call como lo describio Warren ..
END call;
```

La operación auxiliar (interna a la WAM) llamada "backtrack" (ver [6] página 100), fue modificada para que actualice los registros "PC", "H", "E", "B", "HB", y "TR" con los valores de "PC_NAF", "H_NAF", "E_NAF", "B_NAF", "HB_NAF", y "TR_NAF", respectivamente.

```
PROCEDURE BackTrack;
BEGIN
      .. codigo de backtrack descripto por Warren ..
      IF N > 0 THEN
       BEGIN
         PC:=PC_NAF;  {Actualiza los registros *_NAF}
         H:=H_NAF;
         E:=E_NAF;
         B:=B_NAF;
         HB:=HB_NAF;
         TR:=TR_NAF;
       END;
END BackTrack;
```

Por último se incluye el código de **end_naf** que es el encargado de restaurar el estado de todos los registro involucrados en la negación por falla.

```
PROCEDURE End_naf;
BEGIN
Falla:=NOT Falla;
N := - 1;
IF N > 0
THEN Recuperar_del_Stack (PC_NAF, H_NAF, E_NAF, B_NAF, HB_NAF, TR_NAF)
END End_naf;
```

5 Conclusiones y desarrollos futuros

Como se explicó anteriormente, la realización de este trabajo es parte de un proyecto de investigación de mayor envergadura: definir una máquina abstracta para *la programación en lógica rebatible* [12, 1], y construir un intérprete para este lenguaje. Actualmente, la WAM ha sido aceptada como un estandar para la implementación de Prolog. Como la programación en lógica rebatible está definida como una extensión de la programación en lógica, la WAM constituye un excelente punto de partida para este proyecto. Este trabajo presenta el primer paso en la construcción de un interprete para programas lógicos rebatibles: describe un compilador para programas lógicos que genera código WAM, y extiende la WAM para poder utilizar la negación por falla.

Los próximos pasos serán extender la WAM para producir una máquina abstracta para programas lógicos rebatibles, y construir un interprete de esta máquina abstracta, utilizando la semántica definida en [12]. Dicha semántica está basada en los conceptos de la *argumentación rebatible* [11, 3, 13] y define cuatro conjuntos de respuestas para un programa lógico rebatible: el de las respuestas positivas, las negativas, las indecisas, y las desconocidas (ver [12] y [1]).

6 Referencias

[1] García A. J. *Una aproximación a la programación en lógica rebatible.* Remitido al 2do. Workshop en Aspectos Teoricos de la Inteligencia Artificial (ATIA'95). Bahía Blanca, Octubre de 1995.

[2] García A. J. *Extensiones de la programación en lógica.* Reporte Técnico GIIA-UNS-1995-4. Universidad Nacional del Sur.

[3] García A. J., Chesñevar C. I., and Simari G. R. *Making Argument Systems Computationally Attractive.* Proc. XIII Int. Conf. of the Chilean Society for Computer Science, October 1993.

[4] García A. J., Chesñevar C. I., and Simari G. R. *Bases de Argumentos: su mantenimiento y revisión.* XIX Conferencia Latinoamericana de Informática. Buenos Aires, Agosto 1993.

[5] Gelfond M. and Lifschitz V. *Logic Programs with Classical Negation.* Proc. of 7th. Int. Conf. on Logic Programming (ICLP) 1990

[6] H. Aït-Kaci Warren's abstract machine, MIT Press, 1991.

[7] Inoue K. *Extended Logic Programming with Default Assumptions.* Proc. of 8th. Int. Conf. on Logic Programming (ICLP) 1991.

[8] Kowalski R. and Sadri F. *Logic Programs with Exceptions.* Proc. of 7th. Int. Conf. on Logic Programming (ICLP) 1990

[9] Lloyd J. W. *Foundations of Logical Programming.* 2nd. edition, Springer-Verlag 1987

[10] Nute Donald, *Basic defeasible logic*, in *Intensional Logics for Programming*, Ed by Luis Fariñas del Cerro, Claredon Press - Oxford (c) 1992.

[11] Simari G. R. and Loui R. P. *A Mathematical Treatment of Defeasible Reasoning and its Implementation.* Artificial Intelligence, 53: 125–157,1992.

[12] Simari G. R. y García A. J. *A Knowledge Representation Language for Defeasible Argumentation.* XXI Conferencia Latinoamericana de Informática. Canela, Brasil, Agosto 1995.

[13] Vreeswijk G. *Studies in Defeasible Argumentation (Ph.D. Thesis).* Vrije University, Amsterdam, Holanda. March 1993.

[14] Warren David H. D. *Compiling Predicate Logic Programs*, Volumenes 1 y 2, D.A.I. Research Reports Nro. 39. Mayo de 1977.

[15] Warren David H. D. *An Abstract Prolog Instruction Set* Thechnical Note 309, SRI International, Menlo Park, CA, October 1983.